**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Komputeralgebra Tanszék**

2D felülnézetes akciójáték

Szerző: Témavezető:

Sámson Bálint Ferenc Magyar Péter

Programtervező informatikus BSc. Egyetemi tanársegéd

**Szombathely, 2025**

**Téma bejelentő címe:** 2D felülnézetes akciójáték

**Téma bejelentő leírása:**

Egy játék, ami felülnézetes, 2D top-down shooter, ahol különböző pályákat kell teljesíteni. Minden pálya befejezése után kinyílik egy ajtó, amerre tovább lehet haladni. A játékban előre elkészített pályák közül véletlenszerűen sorsolódik a következő. Különleges pályák is vannak, melyek a játék történetének előrehaladását segítik. Két pálya között átvezető animáció van.

A játékban található egy pontszámláló, ezt a játékos ellenfelek megölésével tudja növelni. Bónusz pont jár azért, ha dupla, tripla vagy multi ölést szerez a játékos. Egy pálya befejezése után egy szorzó is szerezhető az alapján, hogy milyen gyorsan sikerült teljesíteni.

A játékban az alapvető mozgás a W, A, S, D gombokkal valósul meg. Ezekkel tudunk előre, balra, hátra, jobbra, illetve átlósan mozogni. A játékosnak lehetősége van falakra felkapaszkodni, és ugrálni közöttük. Megtalálható a játékban egy rakétavető fegyver is, amivel a harcoláson kívül a játékos lövéskor egy lökést kap azzal ellentétes irányba, amelyik irányba lő vele.

Összesen 5 fegyvert használhatunk: egy baseball ütőt, egy Glock pisztolyt, egy kétcsövű shotgunt, egy M4 gépkarabélyt és egy rakétavetőt.

A játék elkészítéséhez a Godot játékmotor lesz használva, és a játék kódja GDScriptben lesz írva.

Tartalomjegyzék

[1. Bevezetés 1](#_Toc195615094)

[1.1. Témaválasztás és célkitűzés 1](#_Toc195615095)

[1.2. Fejlesztési környezet és technológiai háttér 2](#_Toc195615096)

[2. Felhasználói dokumentáció 3](#_Toc195615097)

[2.1. Célközönség 3](#_Toc195615098)

[2.2. Rendszerkövetelmények 3](#_Toc195615099)

[2.2.1. Minimális rendszerkövetelmény 3](#_Toc195615100)

[2.2.2. Ajánlott rendszerkövetelmény 3](#_Toc195615101)

[2.2.3. Fontos megjegyzések 3](#_Toc195615102)

[2.3. A játék indítása és kezelése 4](#_Toc195615103)

[2.3.1. Telepítés és indítás 4](#_Toc195615104)

[2.4. Felhasználói felület elemei 4](#_Toc195615105)

[2.4.1. Főmenü 4](#_Toc195615106)

[2.4.2. Játék közbeni felület (HUD) 5](#_Toc195615107)

[2.4.3. Halál képernyő (Death screen) 6](#_Toc195615108)

[2.4.4. Szint teljesítés képernyő (Level completed screen) 7](#_Toc195615109)

[2.4.5. Játék befejezés képernyő (Game completed screen) 8](#_Toc195615110)

[2.4.6. Átvezetési animációk (Transition animations) 9](#_Toc195615111)

[2.4.7. Szünet menü (Pause menü) 10](#_Toc195615112)

[2.4.8. irányítás 11](#_Toc195615113)

[2.5. Játékmenet alapjai 12](#_Toc195615114)

[2.5.1. Mentés és betöltés 13](#_Toc195615115)

[3. Fejlesztői dokumentáció 14](#_Toc195615116)

[3.1. Megoldási terv 14](#_Toc195615117)

[3.1.1. Rendszer architektúra 14](#_Toc195615118)

[3.1.2. Osztályszerkezet 17](#_Toc195615119)

[3.1.3. Állapotgépek 21](#_Toc195615120)

[3.1.4. Modul struktúra 25](#_Toc195615121)

[3.1.5. Felhasználói felület terve 34](#_Toc195615122)

[3.2. Megvalósítás részei 36](#_Toc195615123)

[3.2.1. GDScript nyelvi elemeinek használata 36](#_Toc195615124)

[3.2.2. Adatábrázolási döntések 36](#_Toc195615125)

[3.2.3. Godot beépített funkcióinak használata 37](#_Toc195615126)

[3.2.4. Scene és script fájlók implementációja 37](#_Toc195615127)

[3.2.5. Tesztelés 46](#_Toc195615128)

[4. Összefoglalás és további fejlesztési lehetőségek 47](#_Toc195615129)

[5. Irodalomjegyzék 48](#_Toc195615130)

[6. Melléklet 49](#_Toc195615131)

# Bevezetés

A videójátékok manapság egyre inkább a „minél több, annál jobb” elvet követik. Tökéletes példa erre a Cyberpunk 2077 – egy gyönyörű, izgalmas világ veszi körül a játékost, de annyira túlzsúfolt, hogy könnyű elveszíteni benne a fonalat. Három óra játékmenet után fogalmam sem volt, hogy egyáltalán melyik a fő sztorivonal. Ezek a játékok próbálnak minden téren tökéletesek lenni sok tartalommal, de talán pont emiatt rontanak a játékélményen.

Ezzel szemben egyre értékesebbé válnak az egyszerű játékok, amelyek kifinomult mechanikára építenek, könnyű játékmenetet kínálnak, és nem kényszerítik a játékost arra, hogy órákat töltsön a számítógép előtt ahhoz, hogy jól érezze magát. Ezek a játékok visszanyúlnak elődjeikhez, ahol a játékélményt a könnyen tanulható, viszont nehezen tökéletesíthető játékmechanikák alkották. A szakdolgozatomat pontosan ezekre az elvekre alapoztam: egy olyan felülnézetes akciójáték készítése volt a célom, amely egyszerűen elsajátítható, és pont elegendő mélységgel rendelkezik ahhoz, hogy lekösse a játékosokat.

## Témaválasztás és célkitűzés

A videójátékok gyerekkorom óta meghatározó szerepet játszanak az életemben. Az egyik legmeghatározóbb játékélményem a Hotline Miami volt, ami egy intenzív, gyors tempójú 2D felülnézetes akciójáték. Ez a játék a nyers játékmenetével és kihívást jelentő nehézségi szintjével gyorsan elnyerte a tetszésemet. Ez az élmény inspirált arra, hogy szakdolgozatként egy hasonló játékot hozzak létre. A felülnézetes shooter műfaj ideális választásnak bizonyult, mivel lehetőséget adott arra, hogy viszonylag egyszerű eszközökkel komplex és szórakoztató játékélményt hozzak létre.

A játék fejlesztése során több konkrét célt tűztem ki magam elé:

* Jól skálázható kódstruktúra kialakítása, amely lehetővé teszi a játék egyszerű bővítését
* Változatos játékmechanikák: különböző fegyverek, ellenséges navigáció, véletlenszerű pályasorsolás
* Komplett játékélmény létrehozása, kezdve a főmenütől a befejező képernyőig
* A Godot játékmotor adottságainak elsajátítása

## Fejlesztési környezet és technológiai háttér

A projekt megvalósításához a Godot játékmotor 4.3-as verzióját választottam, ami egy nyílt forráskódú, rugalmas játékfejlesztő környezet. A Godot mellett több meggyőző érv is szólt: teljesen ingyenes, kiváló a dokumentációja és node-alapú architektúrát használ. Az utóbbi lehetővé teszi a játékelemek moduláris és hierarchikus szervezését.

Minden játékelem (scene) különböző típusú node fából áll, amelyek mindegyike speciális funkcióval rendelkezik. Ez a megközelítés rendkívül rugalmas és újrafelhasználható komponensek létrehozását teszi lehetővé, ami jelentősen megkönnyítette a fejlesztést.

A Godot másik jelentős előnye a beépített szignál rendszer, amely eseményvezérelt programozást tesz lehetővé. Ez a rendszer lehetővé teszi a különböző komponensek közötti kommunikációt anélkül, hogy szoros függőségek alakulnának ki közöttük. Ennek köszönhetően a kód moduláris és könnyen karbantartható marad.

A fejlesztéshez a Godot saját programozási nyelvét, a GDScript-et használtam. Ez a Python-inspirált szkriptnyelv kifejezetten játékfejlesztésre lett optimalizálva. A nyelv szintaxisa könnyen tanulható, mégis erőteljes nyelvi eszközöket kínál. A nyelv támogatja az objektumorientált programozást, a szigorú típusozást (opcionálisan), és számos beépített függvényt kínál játékfejlesztési feladatokhoz.

# Felhasználói dokumentáció

## Célközönség

A játék elsősorban a pörgős akciójátékok kedvelőinek készült, különösen azoknak, akik szeretik a klasszikus felülnézetes shooter műfaj egyszerűségét és intenzitását. A 16+ korosztálynak ajánlott a játék véreffektjei és az erőszakos játékmenet miatt. Ideális választás lehet azoknak, akik kevesebb idővel rendelkeznek, és 15-30 percet játszanának.

## Rendszerkövetelmények

Mivel egy 2D játékról van szó, a rendszerkövetelmények viszonylag alacsonyak a modern 3D játékokhoz képest, azonban a játék fizikai szimulációi bizonyos minimális teljesítményt igényelnek.

### Minimális rendszerkövetelmény

**Operációs rendszer:** Windows 7 SP1 / Windows 8.1 / Windows 10 / Windows 11

**Processzor:** Dual-core CPU, 2.0 GHz vagy gyorsabb

**Memória:** 4GB RAM

**Grafika:** DirectX 11 kompatibilis videokártya

**Tárhely:** 200 MB szabad tárhely

### Ajánlott rendszerkövetelmény

**Processzor:** Quad-core CPU, 2.5 GHz vagy gyorsabb

**Memória:** 8GB RAM

**Grafika:** Dedikált videokártya 2 GB VRAM-mal

### Fontos megjegyzések

A hardverkövetelményeken kívül szükséges egér és billentyűzet bemeneti eszköznek. Hálózati követelmény nincs, mivel a játék teljes egészében offline működik. A játék teljesítménye nagyban függhet a háttérben futó alkalmazásoktól. Alacsonyabb specifikációjú rendszereken a játék működhet, de a teljesítmény csökkenhet, különösen ha sok ellenség van egyszerre a pályán.

## A játék indítása és kezelése

### Telepítés és indítás

A játék telepítése rendkívül egyszerű, mivel önálló futtatható állományként (executable) kerül terjesztésre. A telepítési folyamat a következő:

1. Töltse le a játékot.
2. Nincs szükség telepítésre – egyszerűen csomagolja ki a letöltött fájlt egy tetszőleges mappába, és indítsa el a futtatható állományt.

A játék indításakor megjelenik a főmenü, ahonnan a játék elindítható.

## Felhasználói felület elemei

A játék felhasználói felülete több különböző komponensből épül fel, amelyek a játékélmény egy-egy aspektusát támogatják, a főmenütől kezdve a játék közbeni HUD-on át a különböző állapotképernyőkig.

### Főmenü

A főmenü a játék indításakor jelenik meg, és annak kezdőpontjaként szolgál.



. ábra: Főmenü

Elemei:

* Játékcím: A képernyő középső részén található („Counterfeit” felirat).
* Start gomb: A játék elindítását teszi lehetővé. Amikor ez a gomb fókuszban van, a „>Start<” formázás jelzi a kijelölést.
* Quit gomb: Kilépést biztosít a játékból. Fókuszban a „>Quit<” formázás jelzi a kijelölést.

A főmenüben a fel és le nyilakkal lehet navigálni a gombok között, az Enter vagy Space billentyűkkel pedig megnyomni őket. A játék a főmenűben alapértelmezetten a Start gombra helyezi a fókuszt.

### Játék közbeni felület (HUD)

A játék közbeni felület (Heads-Up Display, HUD) több, a képernyő különböző részein elhelyezett információs elemet tartalmaz. Ezek a folyamat során folyamatosan láthatók és visszajelzést adnak a játékos állapotáról.



2. ábra: Játék közbeni felület (HUD)

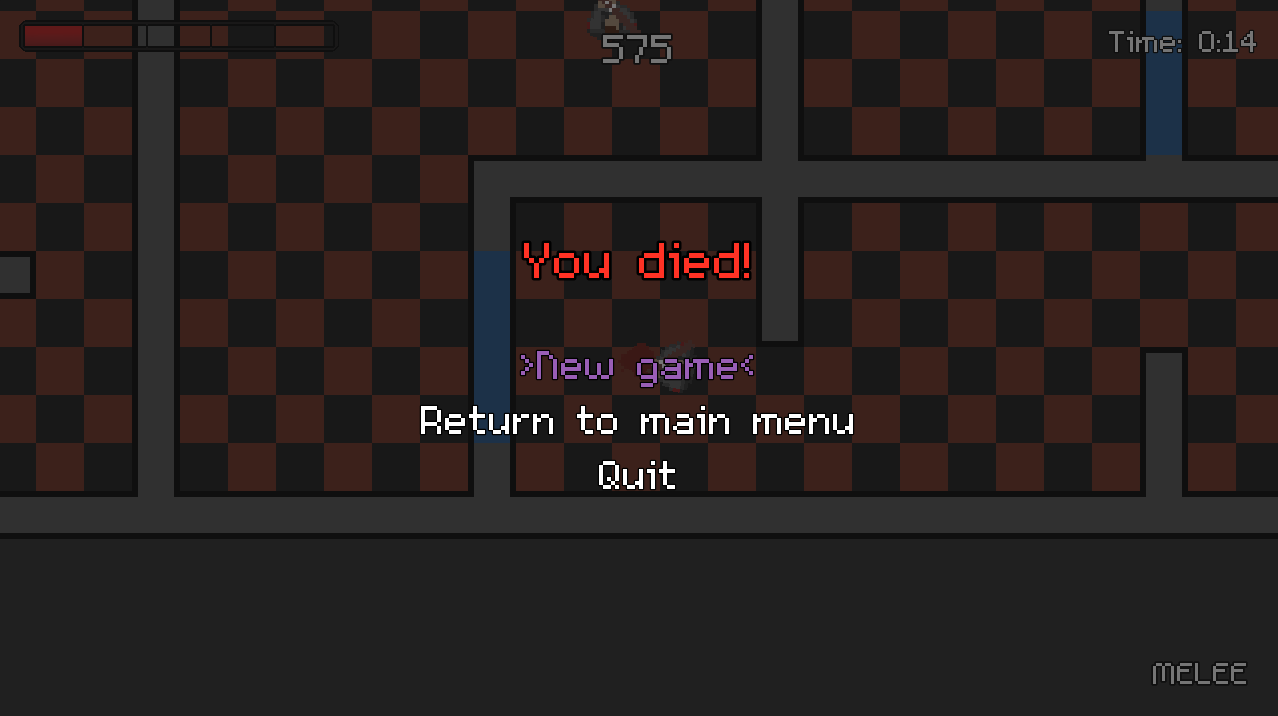
Elemei:

* Életerő kijelző (Health Display): A képernyő bal felső sarkában található. Vizuálisan jelzi a játékos aktuális életerejét 5 különböző állapottal (5 teljes sáv 100% életerőnél, 4 sáv 80% életerőnél, és így tovább).
* Lőszer kijelző (Ammo Display): A képernyő jobb alsó sarkában található. Formátuma „X / Y”, ahol X az aktuális lőszermennyiség, Y pedig a maximális lőszerkapacitás. A baseball ütő esetében „MELEE” felirat jelenik meg. Újratöltés során a lőszer mutató alatt egy újratöltési folyamatjelző jelenik meg, amely vizuálisan jelzi az újratöltés előrehaladását.
* Pontszám kijelző (Score Display): A képernyő felső középső részén található, és az aktuális pontszámot mutatja. Alatta ideiglenesen megjelennek killstreak jelzések („Single kill!”, „Double kill!”, „Triple kill!”, „MULTI KILL!”), amelyek a sorozatos ellenségek legyőzését jelzik és jutalmazzák.
* Idő kijelző (Timer Display): A képernyő jobb felső sarkában található, az aktuális pályán eltöltött időt mutatja.

A HUD minden eleme úgy van elhelyezve, hogy jól látható legyen, és ne akadályozza a játékmenetet.

### Halál képernyő (Death screen)

Amikor a játékos karaktere meghal (életereje nullára csökken), megjelenik a halál képernyő. Ez a képernyő szünetelteti a játékot és lehetőséget biztosít a játékosnak új játék indítására, visszatérni a főmenübe és kilépni a játékból.



3. ábra: Halál képernyő (Death screen)

Elemei:

* „You died!” („Meghaltál!”) felirat a képernyő közepén, piros színnel kiemelve
* Három választási lehetőség:
* New game gomb: Új játék indítása. Amikor ez a gomb fókuszban van, a „>New game<” formázás jelzi a kijelölést.
* Return to main menu gomb: Visszatérés a főmenübe. Amikor ez a gomb fókuszban van, a „>Return to main menu<” formázás jelzi a kijelölést.
* Quit gomb: Kilépés a játékból. Amikor ez a gomb fókuszban van, a „>Quit<” formázás jelzi a kijelölést.

### Szint teljesítés képernyő (Level completed screen)

Amikor a játékos sikeresen teljesít egy pályát, megjelenik a szint teljesítés képernyő, amely összegzi a teljesítményt és lehetőséget ad a következő pályára lépéshez.



4. ábra: Szint teljesítés képernyő

Elemei:

* „Level completed!” felirat
* Befejezési idő (Time): Mennyi idő alatt teljesítette a játékos a pályát
* Sebesség szorzó: Elért időalapú pontszámszorzó
* Pontszám: Az aktuális pontszám a szorzók után
* Continue to next level gomb: Továbblépés a következő pályára.

### Játék befejezés képernyő (Game completed screen)

A játék összes pályájának teljesítése után megjelenik a játék befejezés képernyő, amely összegzi a játékos teljesítményét.



5. ábra: Játék teljesítés képernyő

Elemei:

* „Game is completed!” felirat.
* Végső pontszám (Final score): A játék során szerzett összes pont (szorzók után).
* Teljes játékidő: A teljes játékmenet ideje.
* Három választási lehetőség:
* Main menu gomb: Visszatérés a főmenübe. Amikor ez a gomb fókuszban van, a „>Main menu<” formázás jelzi a kijelölést.
* Quit gomb: Kilépés a játékból. Amikor ez a gomb fókuszban van, a „>Quit<” formázás jelzi a kijelölést.

### Átvezetési animációk (Transition animations)

A játékban két átvezetési animáció található, amelyek a történet előrehaladását jelzik a második és a negyedik pálya teljesítése után.



6. ábra: Átvezetési animáció

Elemei:

* Animált képsorozat, amely automatikusan lejátszódik
* „Press any key to continue” felirat, amely az animáció befejezése után jelenik meg rövid időn belül
* Bármely billentyű megnyomásával a játékos továbbléphet

### Szünet menü (Pause menü)

A játékos az ESC billentyű megnyomásával bármikor megnyithatja a szünet menüt, amely felfüggeszti a játékmenetet és lehetőséget ad a játék folytatására, a főmenübe való visszatérésre vagy a kilépésre.



7. ábra: Szünet menü

Elemei:

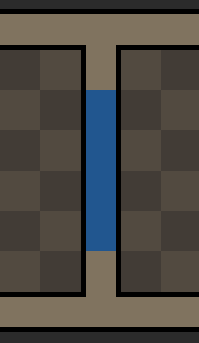
* Resume gomb: Folytatja a játékot onnan, ahol a játékos megállította. Amikor ez a gomb fókuszban van, a „>Resume<” formázás jelzi a kijelölést.
* Return to main menu menu gomb: Visszatérés a főmenübe. Amikor ez a gomb fókuszban van, a „>Return to main menu<” formázás jelzi a kijelölést.
* Quit: Kilépés a játékból. Amikor ez a gomb fókuszban van, a „>Quit<” formázás jelzi a kijelölést.

### irányítás

A játék irányítása az alábbi billentyűzetkiosztással történik:

**Mozgás:**

* W: Előre (felfelé) mozgás
* A: Balra mozgás
* S: Hátra (lefelé) mozgás
* D: Jobbra mozgás
* Space (szóköz): Falugró képesség használata (egy ugrófal közelében)



8. ábra: Ugrófal

**Harc:**

* Bal egérgomb: Lövés/Támadás
* R: Fegyver újratöltése (csak lőfegyverek esetén)

**Általános irányítás:**

* Egér mozgatása: Karakter forgása (a karakter mindig az egérkurzor felé néz)
* ESC: Játék szüneteltetése
* Tetszőleges billentyű: Átvezetési animáció átugrása

A játékos a W, A, S, D billentyűk kombinációjával szabadon mozoghat bármely irányba, miközben az egér pozíciója határozza meg a karakter nézési irányát. Ez a konfigurációs séma lehetővé teszi, hogy a játékos akár hátrafelé mozogva is támadjon/tüzeljen.

## Játékmenet alapjai

A játék célja az összes ellenség legyőzése a pályán, majd a befejező ajtón való áthaladás a következő szintre jutáshoz. A játék öt különböző pályából áll, ezeknek a pályáknak a sorrendje minden játékindítással véletlenszerűen sorsolódik.

A játékos minden pálya elején egy meghatározott fegyverrel indul:

1. Pálya: Baseball ütő (közelharci fegyver)
2. Pálya: Glock 18 (félautomata pisztoly)
3. Pálya: Dupla csövű shotgun (sörétes puska)
4. Pálya: M4 (automata gépkarabély)
5. Pálya: Rakétavető

A képernyő bal felső sarkában található a játékos életereje, amely kezdetben 100 pont. Ha ez nullára csökken, a játékos meghal, és újra kell kezdenie a játékot. A képernyő jobb felső sarkában látható az eltelt idő az aktuális pályán, míg a jobb alsó sarokban az aktuális fegyver lőszere (kivéve a baseball ütő esetében, ahol „MELEE” felirat jelenik meg).

A játékos pontszáma a középső felső részen látható. A pontszám ellenségek legyőzésével növekedik, illetve további bónuszpontok szerezhetők pályák gyors teljesítéséért és killstreak-ekért (több ellenség rövid időn belüli legyőzéséért).

Amikor minden ellenség legyőzésre került egy pályán, a befejező ajtó kinyílik és a játékos továbbhaladhat. A játék teljesítéséhez mind az öt pályát teljesíteni kell egymás után.

### Mentés és betöltés

Mivel a játék pár percen belül teljesíthető, a játék nem rendelkezik mentési funkciókkal. Ha a játékos kilép vagy meghal, a játékot az elejétől kell újrakezdeni.

# Fejlesztői dokumentáció

## Megoldási terv

### Rendszer architektúra

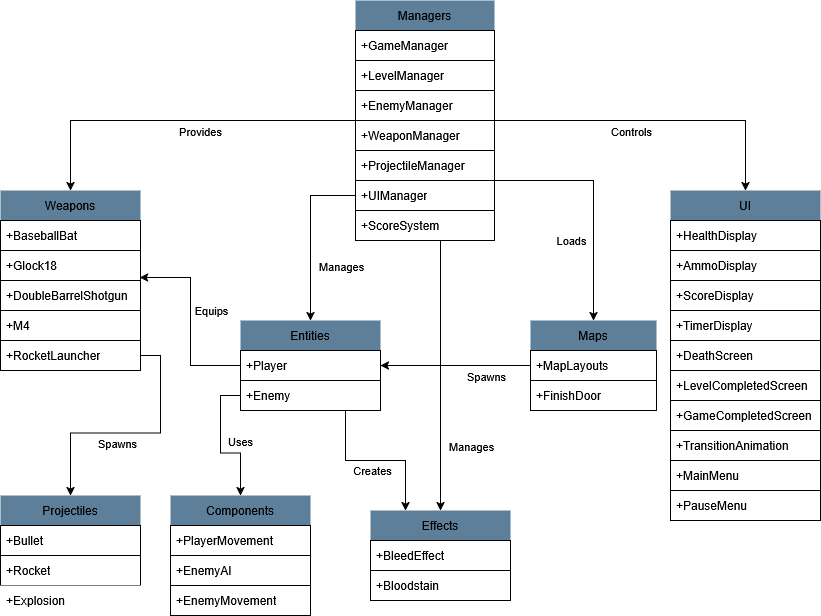
A projekt fejlesztéséhez a Godot játékmotor 4.3-as verzióját használtam, amely egy nyílt forráskódú, könnyen használható fejlesztői környezetet biztosít. A Godot egyik legnagyobb előnye a node-alapú architektúra, amely lehetővé teszi a játék elemeinek hierarchikus szervezését és moduláris kialakítását.

A node-alapú architektúra lényege, hogy minden játékelem (például egy fegyver vagy egy UI elem) különböző típusú node-ok fájából áll, amelyek mindegyike speciális funkcióval rendelkezik. Például a Player node tartalmaz egy AnimatedSprite2D node-ot az animációk kezeléséhez, egy CollisionShape2D node-ot az ütközések kezeléséhez, és egy Camera2D node-ot a játékos követéséhez.

A Godot scene-tree rendszere lehetővé teszi az összetett játékelemek létrehozását egyszerűbb komponensekből. Például a játékban minden fegyver saját scene-t alkot, melyek tartalmazzák a megfelelő animációkat és egyéb tulajdonságokat. Ezek a scene-ek könnyen hozzáadhatók a játékos vagy az ellenfelek node-jaihoz futási időben, amely egyszerűvé teszi a fegyverek cseréjét a játék során.

A játék tervezése során törekedtem a jól elkülöníthető rétegek kialakítására, ezzel segítve a kód karbantarthatóságát. A fő rétegek a következők:

* Manager-ek: Ez a réteg felelős a játék irányításáért. Ide tartoznak a GameManager, LevelManager, EnemyManager, WeaponManager, ProjectileManager, UIManager és a ScoreSystem osztályok. Ezek a menedzserek biztosítják a játék állapotainak megfelelő kezelését.
* Entity-k: Ez a réteg tartalmazza a játék alapvető interaktív elemeit, mint a Player és az Enemy osztályok. Ezek az entitások a játék világában létező objektumok, amelyek viselkedését scriptek irányítják.
* Komponensek: Ide tartoznak az entitások viselkedését kiegészítő vagy módosító elemek, mint a PlayerMovement, EnemyAI, EnemyMovement, valamint a különböző fegyvertípusok (BaseballBat, Glock18, DoubleBarrel, M4, RocketLauncher)
* Projectile-ok: Ez a réteg kezeli a lövedékeket, mint a Bullet, Rocket, illetve a Rocket-hez tartozó Explosion osztályok. Ezek az objektumok a fegyverek által kilőtt lövedékek, melyek kezelik az ütközéseket más objektumokkal.
* UI: Ez a réteg a felhasználói felület elemeit tartalmazza, mint a UI, MainMenu, PauseMenu, DeathScreen, LevelCompletedScreen és a GameCompletedScreen. Ezek a játékos és a játék közötti interakciók vizuális megjelenítéséért felelősek.
* Pályák: Ez a réteg tartalmazza a játék különböző pályáit és a hozzájuk kapcsolódó objektumokat, mint a FinishDoor. A pályák a játék környezetét alkotják, amelyben a játékos és az ellenfelek interakcióba lépnek.



9. ábra: A projekt struktúrája

A Godot egyik jelentős előnye a beépített szignál rendszer, amely eseményvezérelt programozást tesz lehetővé. Ezzel a rendszerrel hozhatjuk létre a különböző komponensek közötti kommunikációt anélkül, hogy szoros függőségek alakulnának ki közöttük.

A játékban számos szignált implementáltam, például:

* enemy\_died: Egy ellenfél legyőzésekor aktiválódik
* game\_started, game\_paused, game\_resumed: A játék állapotváltozásánál aktiválódnak
* map\_ready: Új pálya betöltésekor aktiválódik
* reload\_started: Fegyver újratöltésekor aktiválódik
* score\_changed, killstreak\_updated: Pontszám változásakor aktiválódnak

Ezek a szignálok teszik lehetővé a komponensek közti kommunikációt. Például amikor egy ellenséget legyőz a játékos, az enemy\_died szignál aktiválódik, ami értesíti a GameManager-t, hogy csökkentse az ellenségek számlálóját és a ScoreSystem-et, hogy növelje a pontszámot és frissítse a killstreak számlálót. Mindez úgy történik, hogy az Enemy osztálynak nem kell direkt referenciát tárolnia ezekre az osztályokra.

A fejlesztéshez a Godot saját programozási nyelvét, a GDScript-et használtam, amely egy Python-inspirált szkriptnyelv. Ez a nyelv kifejezetten játékfejlesztésre lett optimalizálva, szintaxisa pedig egyszerűen tanulható, mégis erőteljes nyelvi eszközöket használ.

A GDScript használatának rengeteg előnye volt a projekt elkészítése során:

* Egyszerű szintaxis: A Python-hoz hasonló szintaxis lehetővé tette a kód jó olvashatóságát és a gyors fejlesztést
* Szigorú típusozás (opcionálisan): A projektben gyakran használtam szigorú típusozást
* Beépített vektorfunkciók: A 2D játékban gyakori vektor műveletek egyszerűen használhatók
* Eseményvezérelt programozás támogatása: A szignálrendszer natívan támogatott GDScript-ben, ami megkönnyíti az eseményvezérelt programozást.

### Osztályszerkezet

A játék osztályszerkezete az újrafelhasználható komponensek elvét követi. A fő osztályokat logikusan csoportosítottam, és olyan kapcsolatrendszert alakítottam közöttük, amit könnyebb karbantartani és bővíteni.

#### Fő osztályok áttekintése

1. Entitás osztályok:

Az entitás osztályok a játékban résztvevő karaktereket képviselik.

Player osztály:

* Kezeli a játékos karaktert, annak mozgását, fegyverhasználatát és életpontját
* Feldolgozza a felhasználói bemenetet és megfelelően kezeli
* Lehetővé teszi a fegyverek használatát és cseréjét
* A PlayerMovement komponenssel képes különleges mozgásra (falugró képesség)

Enemy osztály:

* Kezeli az ellenségek viselkedését és állapotait kezeli
* NavigationAgent2D segítségével követi a játékost a legrövidebb útvonalon
* Az EnemyAI és EnemyMovement komponensekkel okos ellenfeleket alkot
* Különböző állapotokban lehet (őrjárat, támadás)

1. Fegyver osztályok:

A játékban több különböző fegyver található, melyek mindegyike sajátos tulajdonságokkal rendelkezik.

BaseballBat osztály:

* Közelharci fegyver
* Támadási területtel és animációval rendelkezik

Glock18 osztály:

* Félautomata lőfegyver korlátozott mennyiségű lőszerrel

DoubleBarrelShotgun osztály:

* Kétlövetű sörétes puska, amely 6 lövedéket lő ki egyszerre, szórással

M4 osztály:

* Automata gépkarabély, amely folyamatosan lő, amíg a lövés gomb le van nyomva
* Nagyobb tárkapacitással rendelkezik, viszont kisebb a sebzése

RocketLauncher osztály:

* Rakétavető, melynek rakétái becsapódáskor felrobbannak
* A játékos visszalökést kap tüzeléskor ellentétes irányba
* A rakéták közvetlen és területi sebzést is okoznak

1. Lövedék osztályok:

A lövedékek a lőfegyverek által kilőtt objektumok.

Bullet osztály:

* Egyenes vonalban haladó lövedék
* Ütközéskor sebzést okoz

Rocket osztály:

* Egyenes vonalban haladó rakéta
* Ütközéskor (vagy az időkorlát elérésekor) felrobban

Explosion osztály:

* A rakéta robbanását kezeli
* Területi sebzést okoz, akár több ellenségnek is

1. Effekt osztályok:

Vizuális effektek a játékélmény növelése érdekében.

BleedEffect osztály:

* Sebződéskor megjelenő vérzés effekt
* Követi a sebzést elszenvedő entitást

Bloodstain osztály:

* Entitás legyőzésekor megjelenő vérfolt effekt, ami a pályán marad

1. Manager osztályok:

A játék központi irányítását a különböző Manager osztályok végzik, melyek mindegyike egy specifikus feladatkörért felelős.

GameManager osztály:

* Kezeli a játék kezdését, újraindítását, szüneteltetését és befejezését
* Regisztrálja és számontartja az ellenségeket
* Szignálokat küld ki a játék állapotváltozásairól

LevelManager osztály:

* A pályák betöltéséért és váltásáért felelős
* Véletlenszerű sorrendben tölti be a pályákat
* Kezeli a játék idő számlálóját és a pályák teljesítési statisztikáit
* A játékos pozicionálást végzi új pálya betöltésekor

EnemyManager osztály:

* Az ellenségek létrehozását és kezelését végzi
* A pályákon elhelyezett spawnpoint-ok szerint helyezi el az ellenfeleket
* Nyilvántartja az aktív ellenségeket

WeaponManager osztály:

* A fegyverek létrehozását és kiosztását végzi
* Az aktuális pálya alapján adja a megfelelő fegyvert a játékosnak és az ellenségeknek

ProjectileManager osztály:

* A lövedékek (bullet, rocket) és robbanások létrehozását kezeli

UIManager osztály:

* A felhasználói felület kezeléséért felelős
* A különböző képernyők és menük megjelenítését és elrejtését kezeli
* A játékos állapotának megjelenítését biztosítja a HUD-on

ScoreSystem osztály:

* A pontszám számítását és nyilvántartását végzi
* Kezeli a killstreak-et és a szorzókat

#### Adatáramlás az osztályok között

A játékban az adatok áramlása főként szignálok segítségével történik, melyek biztosítják a komponensek közötti kapcsolatot. Néhány kiemelt adatáramlási útvonal:

1. Input -> Player -> Weapon -> ProjectileManager:

* A játékos inputot ad (lövés gomb)
* A Player ezt feldolgozza és továbbítja az aktuális fegyvernek
* A fegyver a ProjectileManager-en keresztül létrehozza a megfelelő lövedéket

1. Enemy -> GameManager -> ScoreSystem:

* Amikor egy ellenség meghal, szignált küld a GameManager-nek
* A GameManager csökkenti az ellenségek számát és továbbítja a jelet a ScoreSystem-nek
* A ScoreSystem frissíti a pontszámot és a killstreak számlálót
* A UIManager frissíti a pontszámot és a killstreak-et a HUD-on

1. LevelManager -> EnemyManager -> WeaponManager:

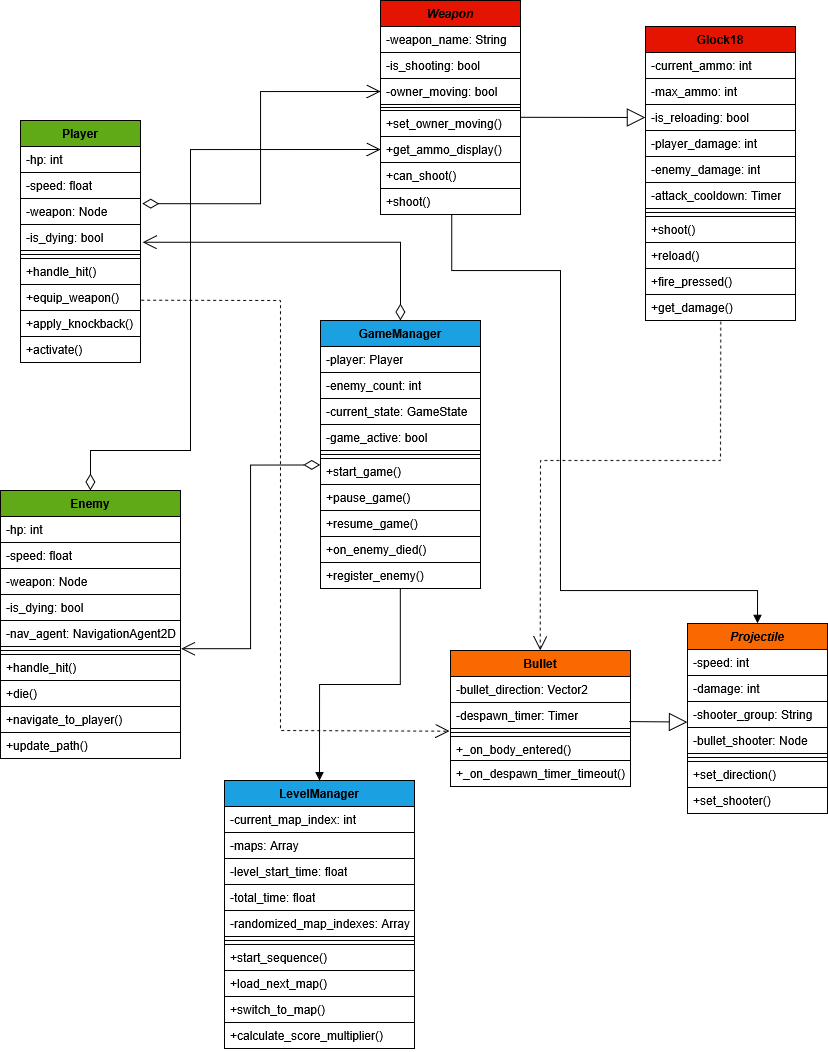
* A LevelManager betölti az új pályát
* Értesíti az EnemyManager-t, hogy helyezze el az ellenségeket
* Az EnemyManager egy ellenség létrehozásakor lekéri a megfelelő fegyvereket a WeaponManager-től
* A WeaponManager az aktuális pálya száma szerint megfelelően választ fegyvert

1. GameManager -> UIManager:

* A játék állapotváltozásainál a GameManager értesíti a UIManager-t
* A UIManager megjeleníti/elrejti a megfelelő felhasználói felületet

#### Osztálydiagram

Az alábbi UML osztálydiagram a játék legfontosabb osztályait és azoknak kapcsolatát ábrázolja. Ez egy egyszerűsített ábra, nem tartalmazza az összes osztályt és metódust a jobb átláthatóság érdekében.



10. ábra: A legfontosabb osztályok és kapcsolataik

### Állapotgépek

A játék működéséhez fontos a különböző objektumok állapotainak nyilvántartása és kezelése. Ehhez állapotgépeket implementáltam, amelyek minden pillanatban egyértelműen meghatározzák egy adott elem viselkedését. A projektben több fontos állapotgép is működik, ezek meghatározzák a játék dinamikáját.

#### Játék állapotai

A játék állapotait a GameManager osztály kezeli, mely három fő állapotot különböztet meg:

* MAIN\_MENU: A játék kiindulópontja, a főmenü megjelenítése
* PLAYING: Ez az aktív játékállapot, amikor a játékos irányítja a karaktert
* PAUSED: Szüneteltetett állapot, ilyenkor a játékmenet fel van függesztve

Az állapotok közötti átmenetek:

* MAIN\_MENU -> PLAYING: A „Start” gomb megnyomásával (start\_game() függvény meghívásával)
* PLAYING -> PAUSED: Az „ESC” billentyű megnyomásával (pause\_game() függvény meghívásával)
* PAUSED -> PLAYING: A „Resume” gomb megnyomásával (resume\_game() függvény meghívásával)
* PAUSED -> MAIN\_MENU: A „Return to main menu” gomb megnyomásával (return\_to\_main\_menu() függvény meghívásával)

Az aktuális játékállapot határozza meg a felhasználói felület elemeit, a bemenetek kezelését és az idő múlásának kezelését (szüneteltetett állapotban a játék nem halad előre).

#### Ellenség viselkedésének állapotai

Az ellenség viselkedésének állapotait az EnemyAI osztály kezeli, amely az egyik legösszetettebb állapotgép a játékban. Ez határozza meg, hogyan viselkedjenek az ellenfelek különböző helyzetekben.

Az EnemyAI két fő állapotot használ:

* GUARD -> ATTACK, ami akkor aktiválódik, ha:
* A játékos belép az ellenség érzékelési zónájába, és közvetlen rálátás van (\_on\_player\_detection\_zone\_body\_entered)
* Egy lövedék (bullet, rocket) halad át az ellenség érzékelési zónáján (\_on\_bullet\_detection\_zone\_area\_entered)
* Az ellenség sebződik a játékos által (handle\_hit)
* ATTACK -> GUARD: GUARD állapotba visszaállítás nem történik a kódban, ami azt jelenti, hogy ha egy ellenség egyszer ATTACK állapotba kerül, akkor abban is marad, amíg nem győzik le vagy újra nem indul a pálya

Az ellenség állapota minden frame-ben kiértékelődik a \_physics\_process függvényben, ahol az aktuális állapotnak megfelelő függvény hívódik meg:

* GUARD állapotban: process\_guard\_state()
* ATTACK állapotban: process\_attack\_state()

#### A GUARD állapot részletesebben

A GUARD állapotban az ellenfél az EnemyMovement osztály segítségével függőleges, járőröző mozgást végez. Ez a következő lépésekben történik:

1. Az EnemyMovement inicializálásakor beállítódnak a guard (járőrözési) pontok
2. Az ellenség a két pont között (patrol\_top\_point, patrol\_bottom\_point) mozog fel és le
3. Ha eléri valamelyik pontot, akkor irányt vált a másik pont felé
4. Eközben folyamatosan figyeli, hogy a játékos betér-e az érzékelési zónájába

#### Az ATTACK állapot részletesebben

Az ATTACK állapotban az ellenfél viselkedése komplexebb:

1. Az ellenség a NavigationAgent2D node segítségével útvonalat keres a játékos felé
2. Az ellenség a játékos felé fordul, majd elindul felé
3. Eközben a navigate\_to\_player() függvény folyamatosan frissíti az ellenség mozgási irányát
4. Ha az ellenség megfelelő távolságba ér a játékostól, megáll és lő (baseball ütőt használó ellenségek csak akkor támadnak, ha nagyon közel vannak)
5. Ha a játékos eltávolodik az ellenség megfelelő lövési távolságától, akkor ismét mozogni kezd, amíg újra közel nem kerül

#### State machine diagram az ellenség állapotgépének bemutatására

#### Player állapotai

A játkos karakter állapotait a kód implicit módon kezeli a következő módon:

1. Aktív állapot:

* is\_active: A játékos aktív vagy inaktív

1. Mozgási állapotok:

* idle: A játékos egy helyben áll
* walk: A játékos előre/hátra mozog
* walk\_sideways: A játékos oldalra mozog

1. Életerő állapotok:

* Normál állapot: hp > 0
* Meghalás: is\_dying igaz, ha hp <= 0

A player állapotai a \_physics\_process() és az update\_animation() függvényekben frissülnek a bemenetek, illetve a játék eseményei szerint.

#### Fegyverek állapotai

A játékban a fegyverek szintúgy állapotgépek alapján működnek. A következő fő állapotokat tartalmazzák:

1. Közös fegyver állapotok:

* idle: A fegyver nincs használatban, és a tulajdonos nem mozog
* walk: A tulajdonos mozog a fegyverrel
* is\_shooting: A fegyver lövési állapotban van

1. Fegyverspecifikus állapotok:

* BaseballBat: is\_attacking -> támadási fázisban van
* Glock18, DoubleBarrel, M4, RocketLauncher: is\_reloading -> újratöltési fázisban van

A fegyverek állapotai az update\_animation\_state() függvényben frissülnek, amely beállítja a megfelelő animációt az aktuális állapot szerint.

### Modul struktúra

A játék moduláris felépítése lehetővé teszi a komponensek függetlenségét és a könnyű karbantarthatóságot. A modulokat logikai csoportokba rendeztem a funkcionalitásuk alapján.

#### Manager modulok

A manager modulok a játék magas szintű irányításáért felelősek:

**GameManager modul:**

A játék központi irányító modulja, amely a játékciklust kezeli és összehangolja a többi manager működését.

* Fő funkcionalitások:
* Játék állapotainak kezelése (főmenü, játék, szüneteltetés
* Játékmenet indítása, szüneteltetése, újraindítása
* Ellenségek nyilvántartása, legyőzésük kezelése
* Cleanup műveletek végrehajtása
* Kapcsolódások:
* Együttműködik az összes többi manager-rel
* Kezeli a játékos és ellenség objektumokat
* Kommunikál a UI manager-rel a játékállapot változásainak megjelenítéséhez

**LevelManager modul:**

Ez a modul felel a pályák betöltéséért, a játékos elhelyezéséért és a pályák közötti átmenetekért.

* Fő funkcionalitások:
* Pályák véletlenszerű sorrendben való betöltése
* Játékos pozicionálása a pályán
* Időszámláló kezelése pályánként
* Pálya befejezésének és átmenetének kezelése
* Pontszám szorzók számítása a teljesítési idő alapján
* Kapcsolódások:
* Együttműködik a FinishDoor objektummal a pálya befejezésének érzékeléséhez
* Kommunikál az EnemyManager modullal az ellenségek elhelyezéséhez
* Jelzéseket küld a UIManager modulnak a megfelelő képernyők megjelenítéséhez

**EnemyManager modul:**

Az ellenségek létrehozásáért és kezeléséért felelős modul.

* Fő funkcionalitások:
* Ellenségek létrehozása a pálya spawnpoint-jai alapján
* Megfelelő fegyverek kiosztása az ellenségeknek
* Ellenségek nyilvántartása és eltüntetése
* Ellenség legyőzésének továbbítása a GameManager modulnak
* Kapcsolódások:
* Együttműködik a WeaponManager modullal a megfelelő fegyver kiosztásához
* Kommunikál a LevelManager modullal pályaváltáskor
* Létrehozza és kezeli az Enemy objektumokat

**WeaponManager modul:**

A fegyverek kezeléséért és kiosztásáért felelős modul.

* Fő funkcionalitások:
* Fegyver példányok létrehozása
* Megfelelő fegyver kiválasztása a pálya sorszáma alapján
* Fegyverek hozzárendelése a játékoshoz és az ellenségekhez
* Kapcsolódások:
* Együttműködik a LevelManager modullal a pálya-specifikus fegyverek meghatározásához
* Létrehozza a fegyver objektumokat (BaseballBat, Glock18, DoubleBarrel, M4, RocketLauncher)

**ProjectileManager modul:**

A kilőtt lövedékek és a robbanások kezeléséért felelős modul.

* Fő funkcionalitások:
* Bullet és Rocket objektumok létrehozása
* Robbanás effektek kezelése
* Lövedékek eltávolítása
* Kapcsolódások:
* Együttműködik a fegyver objektumokkal a lövedékek létrehozásához
* Kezeli a lövedékek és robbanások fizikai interakcióit

**UIManager modul:**

A felhasználói felület kezeléséért és megjelenítéséért felelős modul.

* Fő funkcionalitások:
* Különböző képernyők megjelenítése/eltűntetése (főmenü, halál képernyő, szünet menü)
* Játék közbeni HUD kezelése, UI elemek és játékállapot összehangolása
* Kapcsolódások:
* Együttműködik a GameManager modullal a játékállapot változásainak megjelenítéséhez
* Kommunikál a LevelManager modullal a pályák közötti képernyők megjelenítéséhez
* Kapcsolódik a játékoshoz az életerővel és lőszerrel kapcsolatos információk megjelenítéséhez

**ScoreSystem modul:**

A pontszám számításáért és nyilvántartásáért felelős modul.

* Fő funkcionalitások:
* Pontszámítás ellenségek legyőzésekor
* Killstreak bónuszok kezelése
* Pontszám szorzók alkalmazása pályák teljesítése után
* Kapcsolódások:
* Figyeli a GameManager ellenség legyőzési eseményeit
* Kommunikál a UIManager modullal a pontszám megjelenítéséhez
* Együttműködik a LevelManager modullal a pálya teljesítési szorzók alkalmazásához

#### Játékelemek moduljai

Ezek a modulok a játék entitásait képviselik, és a játékmenet közvetlen résztvevői.

**Player modul:**

A játékos karakteréért felelős modul.

* Fő komponensek:
* player.gd: A játékos alap funkcionalitásának kezelése (mozgás, életerő, fegyverhasználat)
* player\_movement.gd: A játékos speciális mozgásának kezelése (falugrás)
* Funkcionalitások:
* Játékos irányításának és mozgásának kezelése
* Fegyverek használata
* Életerő és sebződés kezelése
* Speciális mozgási képesség (falugrás)

**Enemy modul:**

Az ellenség karakterekért felelős modul.

* Fő komponensek:
* enemy.gd: Az ellenség alap funkcionalitása
* enemy\_ai.gd: Ellenség állapotainak kezelése
* enemy\_movement.gd: Ellenség mozgásának kezelése
* Funkcionalitások:
* Ellenségek viselkedésének és állapotainak kezelése
* Ellenséges navigáció, útvonalkövetés
* Játékos észlelése és követése
* Fegyverhasználat és támadás/lövés

#### Fegyver modulok

A játékban használható különböző fegyverekért felelős modulok.

**BaseballBat modul:**

Közelharci fegyver, amely azonnali sebzést okoz rövid távolságra.

* Funkcionalitások:
* Támadási (ütési) mechanika
* Animáció szinkronizálás
* Cooldown kezelés

**Glock18 modul:**

Félautomata lőfegyver közepes lőszerkapacitással.

* Funkcionalitások:
* Lövési mechanika
* Újratöltés kezelése
* Lőszer nyilvántartása

**DoubleBarrelShotgun modul:**

Kétcsövű sörétes puska, amely több lövedéket lő ki egyszerre, szórással.

* Funkcionalitások:
* Pellet lövési mechanika (több lövedék egyszerre)
* Szórás mechanika
* Újratöltés kezelése
* Lőszer nyilvántartása

**M4 modul:**

Automata gépkarabély, amely folyamatosan tüzel, amíg a tüzelés gomb le van nyomva.

* Funkcionalitások:
* Automata lövési mechanika
* Szórás mechanika
* Újratöltés kezelése
* Lőszer nyilvántartása

**RocketLauncher modul:**

Rakétákat tüzelő fegyver, amely direkt és terület alapú sebzést is okoz. A játékos a lövés irányával ellentétes visszalökést kap.

* Funkcionalitások:
* Robbanó lövedékek
* Direkt és terület alapú sebzés
* Játékos visszalökés (knockback) mechanika
* Újratöltés kezelése
* Lőszer nyilvántartása

#### Lövedék modulok

A fegyverek által kilőtt lövedékek viselkedéséért felelős modulok.

**Bullet modul:**

Alapvető lövedék, amely egyenes vonalban halad.

* Funkcionalitások:
* Egyenes vonalú mozgás
* Ütközések kezelése
* Sebzés találat esetén

**Rocket modul:**

Rakéta, amely becsapódáskor felrobban. Direkt és területi sebzést is tud okozni.

* Funkcionalitások:
* Egyenes vonalú mozgás
* Robbanás létrehozása ütközéskor
* Közvetlen találati sebzés és területi sebzés

**Explosion modul:**

A rakéta robbanásáért felelős modul.

* Funkcionalitások:
* Területi sebzés
* Robbanás vizuális megjelenítése
* Találatok nyilvántartása (egy ellenfél csak egyszer sebződhet egy robbanástól)

#### Effekt modulok

Vizuális effektekért felelős modulok.

**BleedEffect modul:**

Sebződés vizuális jelzése.

* Funkcionalitások:
* Animáció lejátszása sebződéskor
* Karakter követése animáció lejátszása közben

**Bloodstain modul:**

Halál utáni vérfolt megjelenítése.

* Funkcionalitások:
* Animáció lejátszása halálkor
* Vérfolt rögzítése a pályán halál pozíciójában

#### UI modulok

A felhasználói felület megjelenítéséért felelős modulok.

**HUD (Heads-Up Display) modul:**

Játékmenet közbeni információk megjelenítéséért felel.

* Fő komponensek:
* health\_display.gd: Életerő megjelenítése
* ammo\_display.gd: Lőszer megjelenítése
* score\_display.gd: Pontszám és killstreak megjelenítése
* ui.gd: A HUD elemek összekapcsolása
* Funkcionalitások:
* Játékos életerejének grafikus megjelenítése
* Lőszer kijelzése és újratöltési folyamatjelző
* Pontszám és killstreak információk
* Időszámláló megjelenítése

**Screens modul:**

Különböző képernyőkért és menükért felelős modul.

* Fő komponensek:
* main\_menu.gd: Főmenü
* pause\_menu.gd: Szünet menü
* death\_screen.gd: Halál utáni képernyő
* level\_completed\_screen.gd: Pálya teljesítési képernyő
* game\_completed\_screen.gd: Játék befejezés képernyő
* transition\_animation.gd: Átvezetési animációk
* Funkcionalitások:
* Játékállapot váltások kezelése
* Navigáció a menüpontok között
* Játékstatisztikák megjelenítése
* Történet előrehaladásának jelzése

#### Pálya modulok

A játék környezetéért és pályaelemeiért felelős modulok.

**MapLayouts modul:**

A játék pályáit tartalmazó modul.

* Funkcionalitások:
* Falak és akadályok elhelyezése
* Spawn pontok (Player és Enemy) definiálása
* Befejező ajtó (FinishDoor) elhelyezése

**FinishDoor modul:**

A pálya befejezését jelző ajtó modul.

* Funkcionalitások:
* Nyitott és zárt állapotok kezelése és nyitási animáció lejátszása
* Játékos átlépésének érzékelése és következő pályára lépés

#### Modulok közötti kommunikáció

A modulok közötti kommunikáció a következő módszerekkel történik:

1. Szignálok (signals): A Godot beépített eseménykezelő rendszere, amely lehetővé teszi a modulok közötti kommunikációt. Például:

* enemy\_died szignál az Enemy-től a GameManager felé
* game\_started, game\_paused szignálok a GameManager-től a többi manager felé
* score\_changed szignál a ScoreSystem-től a UI felé

1. Közvetlen referencián keresztüli metódus meghívások: Néhány modul közvetlen referenciát tárol egy másik modulra, annak metódusait meghívja. Például:

* A GameManager modul tárolja a Player referenciát
* A fegyverek meghívják a ProjectileManager spawn metódusait

1. Globális hozzáférés node-ok elérési útján keresztül: Néhány esetben a modulok a scene fa használatával szerzik meg a szükséges másik modulokat. Például:

* get\_node\_or\_null(„/root/Main/Managers/GameManager”)

### Felhasználói felület terve

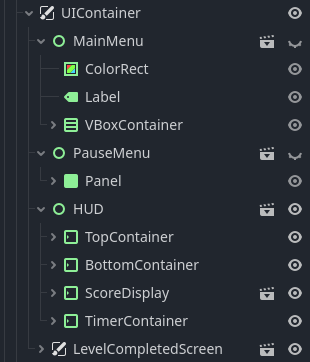
A játék felhasználói felületének implementációja az újrafelhasználhatóságra és a modularitásra törekszik. A UI architektúra kialakításakor a könnyű karbantarthatóság és a játéklogikától való elkülönítés voltak a fő szempontjaim.

#### UI komponens architektúra

A felhasználói felület moduláris felépítését a hierarchikus UI struktúra támogatja. Ez a hierarchia a Godot CanvasLayer rendszerén alapul, amely lehetővé teszi, hogy a különböző elemek a játék világától független rétegként jelenjenek meg. Ezáltal a kamera mozgása és más transzformációk nem befolyásolják a UI elemek pozícióját.

A felület központi irányításáért a UIManager felel. Az osztály nem tárolja el előre betöltve az összes képernyőt, hanem szükség szerint dinamikusan hozza létre és távolítja el őket.

Az alábbi ábrán a UI aktuális struktúrája látható, miközben a játékos a pálya befejező képernyőn van.



12. ábra: UI struktúra a main scene-ben, miközben a játékos a pálya befejező képernyőn van

Jól láthatóak az előre betöltött UI képernyők (MainMenu, PauseMenu, HUD), illetve a dinamikusan betöltött LevelCompletedScreen képernyő.

#### UI elemek technikai megvalósítása

**HealthDisplay:**

Az életerő megjelenítése előre elkészített textúrák váltásával történik, nem pedig dinamikus méretezéssel. Ez a megoldás lehetővé teszi a kinézet kialakítását minden egyes életerő szinthez.

**AmmoDisplay és ReloadProgress:**

Az újratöltési folyamatot egy Tween animációval jelzi a játék, amely változtatja a ProgressBar értékét az újratöltés időtartama alatt. Ez a megoldás lehetővé teszi a folyamatos, nem ugrásszerű változást, és automatikusan kezeli a különböző időtartamú újratöltéseket.

**ScoreDisplay és Killstreak:**

A killstreak rendszer időzítők és animációk segítségével ad dinamikus visszajelzést a játékosnak. A pontszám növekedésekor egy rövid nagyítási animáció jelzi a változást a játékosnak. A killstreak időzítő eközben a streak állapotának törlődéséért felel, ha túl sok idő telik el két ölés között.

**Menü rendszer:**

A menürendszer a Godot beépített fókuszkezelését használja a gombok közötti navigációhoz. Ez egyedi színezéssel és szövegmódosítással egészül ki. A gombok focus\_neighbor\_top és focus\_neighbor\_bottom tulajdonságai lehetővé teszik, hogy a legfelső gombról felfelé lépve a legalsó gombhoz kerüljünk, és a legalsó gombról lefelé lépve a legfelső gombra kerüljünk.

**Képernyők közötti átmenetek:**

A képernyők közötti átmenetek szignál alapú rendszerrel működnek, amely lehetővé teszi a kapcsolást a UI és a játéklogika között. A szintteljesítési képernyők és az átvezetési animációk betöltése igény szerint történik. Az ehhez kapcsolódó logikát a LevelManager kezeli, ezáltal a UI és a játéklogika külön fejleszthető.

**Pause rendszer:**

A játék szüneteltetése a Godot beépített get\_tree().paused tulajdonságának segítségével történik. A PauseMenu process\_mode tulajdonsága „Always” beállítással rendelkezik, amely biztosítja, hogy a játék akkor is aktív maradjon, amikor szüneteltetett állapotban van. Ez teszi lehetővé az interakciót a szünet menüvel.

## Megvalósítás részei

### GDScript nyelvi elemeinek használata

A Godot szkriptnyelve számos tulajdonsággal rendelkezik, ami előnyösnek bizonyult a fejlesztés során. Ide tartozik például az @onready annotáció, amely jelentősen egyszerűsíti a node-referenciák kezelését, mivel csak a scene betöltése után próbálja elérni a megadott node-ot. Ide sorolhatóak még a típusos változók, a setterek és a szignálok definiálása is.

### Adatábrázolási döntések

A projekt megvalósítása során több helyen is dönteni kellett az adatok tárolásáról és kezeléséről:

1. Fegyverek tulajdonságai: A fegyverek tulajdonságait (sebzés, lőszerkapacitás, újratöltési idő, támadási cooldown) közvetlenül a fegyver osztályokban tároltam. Alternatíva lett volna egy központi osztály, ami ezeket kezeli, de a közvetlen tárolás egyszerűbb karbantartást biztosított egy ekkora méretű játéknál.
2. Pályakezelés: A pályákat külön scene fájlokként kezeltem, melyeket a LevelManager dinamikusan tölt be, majd ad hozzá a main scene LevelContainer node-jához. A pályák sorrendjét egy randomizált tömb tárolja.
3. Ellenség AI állapotok: Az ellenség állapotait enum segítségével tároltam, ezzel biztosítva az átláthatóságot.
4. Pontszám rendszer: A pontszámot és a killstreak-et külön osztályban kezeltem (ScoreSystem), így egy helyen történik az adatok tárolása és az időzítők kezelése.

### Godot beépített funkcióinak használata

A Godot játékmotor számos beépített funkcióját használtam a játék készítése során:

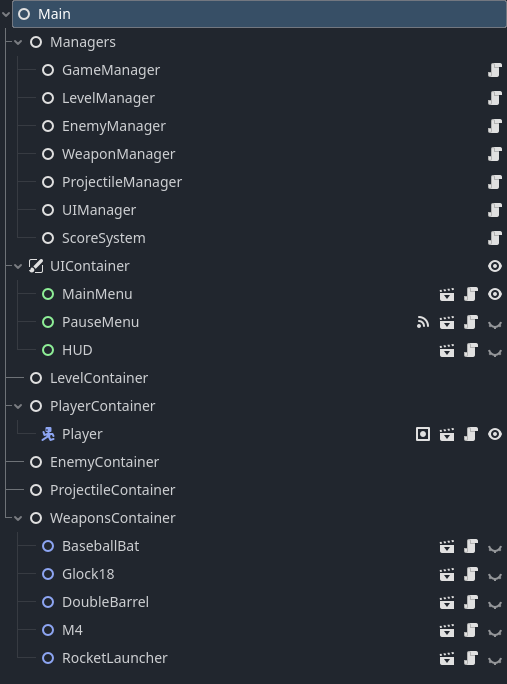
1. Node felépítés: A játék elemeinek hierarchikus felépítése (például a player tartalmazza a fegyvert, a fegyver pedig a töltés idejét) tisztább kódstruktúrát eredményez.
2. Szignál rendszer: A komponensek közötti kapcsolatokhoz szignálokat használtam.
3. NavigationAgent2D: Az ellenségek útvonaltervezéséhez a beépített NavigationAgent2D node-ot használtam, amely hatékony A\* algoritmust segítségével keresi az optimális útvonalat.
4. Tween animációk: UI animációkhoz a Tween rendszert használtam.

### Scene és script fájlok implementációja

#### Main scene (main.tscn)

A játék irányításáért felelős scene, amely hierarchikus node struktúrát használ a különböző feladatok elválasztására.

A Managers, PlayerContainer és WeaponsContainer Node-ok már előre betöltött scene-eket tartalmaznak, míg a LevelContainer, EnemyContainer és ProjectileContainer Node-ok dinamikusan töltik be a scene-jeiket. A UIContainer kivételt képez, hiszen tartalmazza a MainMenu-t, PauseMenu-t és HUD-ot, viszont dinamikusan is tölt be scene-eket a játék aktuális állapota szerint (DeathScreen, LevelCompletedScreen, GameCompletedScreen).



13. ábra: A main scene felépítése

#### Player komponensek

**Player scene (player.tscn) és player.gd:**

A player scene CharacterBody2D típusú node-ot használ, amely több beépített fizikai művelettel is rendelkezik:

* A \_physics\_process() függvény kezeli a W, A, S, D bemeneteket az Input.is\_action\_pressed() segítségével
* A velocity változót használja a mozgás és a visszalökés fizikájához
* A move\_and\_slide() beépített függvény, amely a karakter mozgásáért felelős

**player\_movement.gd:**

Ez a szkript felel a player karakterének falugró képességéért. Ez az alábbi módon valósul meg:

* A legközelebbi fal megkereséséhez RayCast2D node-ot használ
* Az ugrófalakat a find\_all\_nodes\_of\_type() függvény segítségével keresi a TileMapLayer-ek között
* Az érkezési pozíció akadálymentességét a PhysicsRayQueryParameters2D segítségével ellenőrzi

#### Fegyver komponensek

**Baseball ütő scene (baseball\_bat.tscn) és baseball\_bat.gd:**

* Két AnimatedSprite2D node-ot használ (player és enemy)
* Egy Area2D node segítségével határozza meg a találati zónát (AttackArea)
* Timer node-okat használ a támadási cooldown hosszának meghatározására
* Az ütések nyilvántartásához a szkriptben egy hit\_targets tömböt használ
* Az \_on\_attack\_area\_body\_entered() függvény segítségével reagál a találati zónában levő játékosra/ellenségre

**Glock18 scene (glock18.tscn) és glock18.gd:**

* Marker2D node-ot használ a lövedék kiindulási pozíciójának meghatározásához (EndOfGun)
* Timer node-ok segítségével kezeli az újratöltés és tüzelés cooldown-jait
* Az újratöltés állapotának kommunikálására a reload\_started szignált használja
* A shoot() függvény opcionális paraméterrel rendelkezik (target\_direction), amely ha null marad, akkor az egér pozícióját használja lövési iránynak

**Double barreal shotgun scene (double\_barrel\_shotgun.tscn) és double\_barrel\_shotgun.gd:**

* Marker2D node-ot használ a lövedék kiindulási pozíciójának meghatározásához (EndOfGun)
* Timer node-ok segítségével kezeli az újratöltés és tüzelés cooldown-jait
* Az újratöltés állapotának kommunikálására a reload\_started szignált használja
* A shoot() függvény opcionális paraméterrel rendelkezik (target\_direction), amely ha null marad, akkor az egér pozícióját használja lövési iránynak
* A shoot() függvényben történik a több lövedék létrehozása for ciklussal
* A véletlenszerű szóráshoz a beépített randf\_range() függvényt használja

**M4 scene (m4.tscn) és m4.gd:**

* Marker2D node-ot használ a lövedék kiindulási pozíciójának meghatározásához (EndOfGun)
* Timer node-ok segítségével kezeli az újratöltés és tüzelés cooldown-jait
* Az újratöltés állapotának kommunikálására a reload\_started szignált használja
* A shoot() függvény opcionális paraméterrel rendelkezik (target\_direction), amely ha null marad, akkor az egér pozícióját használja lövési iránynak
* A fire\_button\_held bool változóval számontartja, hogy a játékos nyomva tartja-e a lövés gombot (autofire)
* a \_process() függvény folyamatosan ellenőrzi az autofire állapotot, és automatikusan lő, ha igaz
* kis mértékű, véletlenszerű szórást ad minden egyes kilőtt lövedéknek a beépített randf\_range() függvény segítségével

**Rocket launcher scene (rocket\_launcher.tscn) és rocket\_launcher.gd:**

* Marker2D node-ot használ a lövedék kiindulási pozíciójának meghatározásához (EndOfGun)
* Timer node-ok segítségével kezeli az újratöltés és tüzelés cooldown-jait
* Az újratöltés állapotának kommunikálására a reload\_started szignált használja
* A játékos visszalökéséhez tüzeléskor a parent.apply\_knockback() függvényhívást használja
* két sebzési értéket tárol: player\_damage a direkt sebzéshez és play\_explosive\_damage a területi sebzéshez

#### Enemy komponensek

**Enemy scene (enemy.tscn) és enemy.gd:**

* NavigationAgent2D node-ot használja a játékoshoz vezető legoptimálisabb út meghatározásához
* a legyőzésének kommunikáláshoz az enemy\_died szignált használja

**enemy\_ai.gd:**

* Enum-ot használ az állapotok nyilvántartásához (GUARD, ATTACK állapotok)
* Két különböző Area2D node-ot használ a játékos és a lövedék észrevételéhez
* RayCast2D node-ot használ a játékosra rálátás meghatározásához (line\_of\_sight)
* lerp\_angle() függvény segítségével biztosítja a simább fordulást

**enemy\_movement.gd:**

* Két változót használ a járőrözés pontjaihoz: patrol\_top\_point és patrol\_bottom\_point
* Járőrözéskor a lerp\_angle() függvény segítségével biztosítja a simább fordulást
* A fizikát a velocity változón keresztül módosítja, majd a move\_and\_slide() beépített függvényt használja



14. ábra: Az enemy scene felépítése

#### Lövedék komponensek

**Bullet scene (bullet.tscn) és bullet.gd:**

* Area2D node-ot használ az ütközések érzékeléséhez CollisionShape2D-vel
* A \_process() függvényben számítja a bullet sebességét vektorszorzással
* Tárolja a bullet lövőjének csoportját a shooter\_group változó segítségével
* Beállítja a megfelelő ütközési mask-ot a shooter\_group szerint

**Rocket scene (rocket.tscn) és rocket.gd:**

* Area2D node-ot használ az ütközések érzékeléséhez CollisionShape2D-vel
* A bullet-hez hasonló a mozgása, de becsapódáskor robbanást (explosion-t) hoz létre
* A call\_deferred(„explode”) beépített függvényt használja a fizikai motor limitációja miatt
* Az exploded bool változót használja a robbanás nyilvántartására, ennek segítségével elkerülve a többszörös robbanásokat

**Explosion scene (explosion.tscn) és explosion.gd:**

* Area2D node-ot használ a területi sebzés kezeléséhez
* Egy damaged\_targets tömbben tárolja a már megsebzett ellenfeleket, ezzel elkerülve, hogy egy ellenfél többször is sebződjön egy robbanástól

#### UI komponensek

**HUD scene (ui.tscn) és ui.gd:**

* A HUD egyes részei különálló scene-ek (HealthDisplay, AmmoDisplay, ScoreDisplay)
* A find\_player() függvénnyel keresi meg a játékost a csoportja alapján: get\_tree().get\_nodes\_in\_group(„player”)
* A UI elemei folyamatosan frissülnek a \_process() függvény segítségével

**HealthDisplay scene (health\_display.tscn) és health\_display.gd:**

* Több TextureRect node-ot használ az életerő különböző állapotaihoz
* A megfelelő textúrát az update\_health\_bar(health) függvény jeleníti meg

**AmmoDisplay scene (ammo\_display.tscn) és ammo\_display.gd:**

* ProgressBar node-ot használ az újratöltés jelző megjelenítéséhez
* A create\_tween() beépített függvényt használja az újratöltés jelző animációhoz

**ScoreDisplay scene (score\_display.tscn) és score\_display.gd:**

* Időzítőket használ a killstreak nyilvántartásához
* A \_ready() függvényben csatlakozik a ScoreSystem szignáljaihoz
* A create\_tween() függvényt használja a killstreak jelzések animációihoz

**Menük, halál képernyő és befejezési képernyők:**

* A Button node-ok focus\_neighbor tulajdonságát használják a gombok közti navigáláshoz
* A gombok szövege a fókusz alapján módosul, ezt a \_process() függvény tartja számon

**TransitionAnimation scene (transition\_animation.tscn) és transition\_animation.gd:**

* AnimatedSprite2D node-ot használ az animációk lejátszásához
* Az \_input() beépített függvénnyel figyeli a bemenetet (ha a waiting\_for\_input bool változó igaz lesz)
* Szignált küld az animáció befejezésekor (ha jön bemenet a játékostól az időzítő vége után)

#### Manager komponensek

**game\_manager.gd:**

* Enum-ot használ a játékállapotok tárolására (MAIN\_MENU, PLAYING, PAUSED)
* Az \_input függvényben figyeli az Esc billentyű bemenetet a játék szüneteltetéséhez
* Szignálokat küld főbb eseményekhez: game\_started, game\_paused, game\_resumed és enemy\_killed
* Az on\_enemy\_died() függvénnyel kinyitja az ajtót az aktuális pályán, ha nincs több ellenség

**level\_manager.gd:**

* A find\_existing\_maps() függvény segítségével kigyűjti a pályákat a mappájukból
* A randomize\_map\_order() függvény randomizálja a pályák sorrendjét a beépített shuffle() függvénnyel
* A load() és instantiate() függvényeket használja a pályák dinamikus betöltéséhez
* A switch\_to\_map() és load\_next\_map() függvényekkel feldolgozza a jelenlegi pálya befejezését, majd betölti és beállítja a következőt
* A find\_door() függvény megkeresi a befejezési ajtót az aktuális pályán
* A calculate\_score\_multiplier() függvény kiszámolja a szorzót az aktuális pálya időküszöbje szerint

**enemy\_manager.gd, weapon\_manager.gd és projectile\_manager.gd:**

* Az enemy\_manager a spawn\_enemies\_on\_map() függvény segítségével kigyűjti az aktuális pálya spawn point-jait, majd lerakja az ellenségeket
* A weapon\_manager konstans szótárat használ, amiben a fegyverek scene-jeinek elérési útjait tárolja
* A projectile\_manager preload() változókkal tárolja a lövedékek scene-jeit
* Ezek a manager-ek a \_ready() függvényükben csatlakoznak a megfelelő szignálokhoz

**ui\_manager.gd:**

* A load().instantiate() technikával dinamikusan hoz létre bizonyos képernyőket
* Kezeli a UI elemek láthatóságát a visible tulajdonságuk módosításával

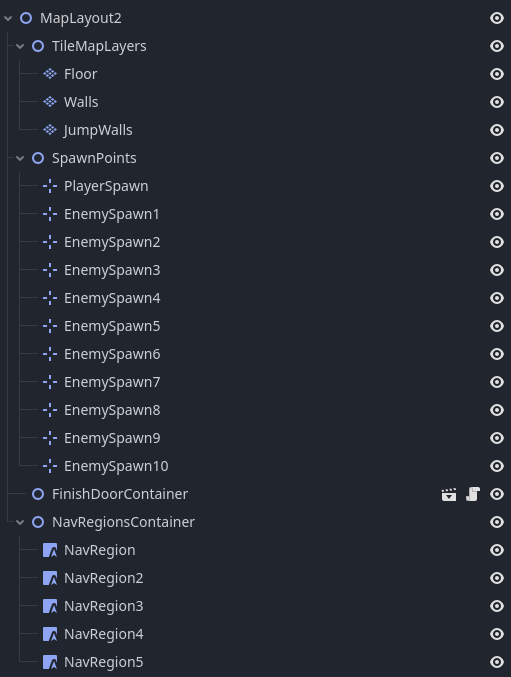
**score\_system.gd:**

* Saját időzítőket hoz létre kódban a killstreak nyilvántartásához
* Szignálokat küld a UI frissítésének jelzésére
* Enum-ot használ a killstreak típusok tárolására (NONE, SINGLE\_KILL, DOUBLE\_KILL, TRIPLE\_KILL, MULTI\_KILL)
* Az időzítők lejáratakor hozzáadja a pontszámhoz a bónuszpontokat

#### Pálya scene-ek (map\_layout\*.tscn)

A játékban öt pálya található, ezek a map\_layout\_1, map\_layout\_2, map\_layout\_3, map\_layout\_4 és a map\_layout\_5 scene-ekben találhatóak.

* A pályák TileMapLayer node-okat használnak a különböző rétegek elkülönítésére: falak, padlók és ugrófalak (Floor, Walls, JumpWalls)
* A SpawnPoints node Marker2D node-okat tartalmaz: itt található a player spawn point-ja (PlayerSpawn) és az ellenségek spawn point-jai is (EnemySpawn1, EnemySpawn2, …)
* Minden pálya tartalmazza a befejezési ajtót (FinishDoorContainer)
* A collision\_layer és collision\_mask tulajdonságok biztosítják, hogy megfelelőek az ütközési szabályok
* Az összes pálya scene tartalmaz egy NavRegionContainer Node2D-t, amely NavigationRegion2D node-okat tartalmaz. Ezek a node-ok határozzák meg a járható területet az enemy NavigationAgent2D node-jának



15. ábra: A map\_layout\_3 scene felépítése

#### FinishDoor scene (finish\_door.tscn)

* A FinishDoor két fő részből áll: Egy Area2D node-ból, amely a játékos áthaladását érzékeli és egy StaticBody2D node-ból, amely a fizikai ütközésért felel, amíg zárva van
* Az ajtó textúráját és animációját egy AnimatedSprite2D node kezeli
* Az ajtó nyitását a GameManager vezérli az on\_enemy\_died() függvényében: ha minden ellenséget legyőzött a játékos, akkor kinyílik az ajtó

### Tesztelés

A projekt készítése során minden funkció implementálása után manuális tesztelést végeztem.

A játékos karakter mozgásának, célzásának és lövésének mechanikáit sok különböző szituációban ellenőriztem. A legtöbb tesztelést igénylő játékmeneti mechanika a falugrás volt, melynél különös figyelmet fordítottam a képesség működésének vizsgálatára.

Az ellenségek útkeresését és navigációját különféle, erre a célra készített pályákon teszteltem, hogy meggyőződjek a megfelelő működésről. A jelenlegi öt pálya mindegyikén ellenőriztem a navigációs területeket, hogy az ellenségek semmiképpen se akadjanak meg sarkokon, vagy sarkokban.

Minden fegyver sebzését, lövési gyorsaságát és animációját teszteltem, hogy megfelelő legyen a működés. Az M4 automata tüzelését és a rakétavető visszalökésének erejét többször is finomítottam.

A felhasználói felület készítésekor törekedtem arra, hogy különböző képernyőméreteken is arányos méretű legyen.

A játék teljes végigjátszása számtalan alkalommal megtörtént, hogy ellenőrizzem a pályaváltások, az átvezetési animációk és a játék befejezésének megfelelő működését. A tesztelés során talált hibákat azonnal javítottam.

# Összefoglalás és további fejlesztési lehetőségek

A szakdolgozat keretében egy 2D felülnézetes top-down shooter játékot terveztem és valósítottam meg a Godot játékmotor használatával. Fő célom egy olyan játék készítése volt, amely visszanyúl a klasszikus akciójátékok alapjaihoz – könnyen tanulható, viszont nehezen tökéletesíthető mechanikák, gyors tempójú játékmenettel.

A fejlesztés során fő szempont volt a könnyű bővíthetőség és a karbantarhatóság. A Godot node-alapú rendszerét használtam a komponensek felépítéséhez, ahol manager osztályok kezelik a központi játéklogikát, míg az egyes node-ok a konkrét játékelemek működését.

Technikai szempontból mindenképp kiemelendő az ellenséges AI, amely útvonaltervezést használ, valamint a falugró képesség, ami további mechanikai elemet ad a játékmenethez. A UI rendszert modulárisra terveztem, dinamikusan betöltött képernyőkkel.

A projektem bemutatja, hogyan valósítható meg egy viszonylag egyszerű játékkoncepció jól strukturált kóddal. Az elkészült játék teljesíti a kezdeti célt: olyan játékélményt teremt, amely egyszerű, mégis lebilincselő.

Több olyan terület is van, ahol a játék a jövőben bővíthető lenne a moduláris felépítésnek köszönhetően. A legkönnyebb az új pályák létrehozása, hiszen elég csak elkészíteni őket a jelenlegi pályák felépítésének mintájára, és kibővíteni a kódot az új pálya elérési útjával.

A projekt egyszerűen kiegészíthető új, egyéni mechanikákkal rendelkező fegyverek hozzáadásával és több ellenségtípussal is, melyek mind növelnék a játékmenet változatosságát. Kerülhet a játékba egy mentési rendszer is sok pálya esetén, így a játékos akármelyik pálya teljesítése közben kiléphet, majd folytathatja onnan, ahol abbahagyta.

Ezek a fejlesztési lehetőségek egyszerűen megvalósíthatók, hiszen a játék jelenlegi állapota egy biztos kiindulópontot biztosít.

# Irodalomjegyzék

1. Godot dokumentáció: <https://docs.godotengine.org/en/4.3/> (Utolsó hozzáférés: 2025. április 15.)
2. Godot játékfejlesztési útmutató és példák: <https://godottutorials.com/> (Utolsó hozzáférés: 2025. április 15.)
3. Vampire Survivors típusú játék elkészítési útmutató Godot 4-ben: <https://www.youtube.com/playlist?list=PLtosjGHWDab682nfZ1f6JSQ1cjap7Ieeb> (Utolsó hozzáférés: 2025. április 17.)
4. Navigációs rendszer és ellenség útvonalkeresése Godot-ban: <https://youtu.be/aW4Oa4dyXA?si=8VdNzwNOpzHCbgWd> (Utolsó hozzáférés: 2025. április 17.)